SYSTEMS AND METHODS FOR A LARGE OBJECT INFRASTRUCTURE IN A DATABASE SYSTEM

FIELD OF THE INVENTION

[0001] The present invention generally relates to the field of database systems and, more specifically, to systems and methods for providing a large object infrastructure that can be used for reimplementation of large object data types (text, ntext, and image), to support large user-defined types, as well as other new data types such as XML.

BACKGROUND OF THE INVENTION

[0002] In database storage platforms, including previously released versions of Microsoft's SQL Server, small value objects are copied without much concern for the overhead involved because such overhead costs are relatively nominal. However, for large-value objects, the overhead costs are much more significant and can negatively impact overall performance.

[0003] Heretofore large objects have been stored using large object (LOB) data types—text, ntext, and image data types—to overcome the size limitations of varchar, nvarchar, and varbinary types for values too large for these data types that are best suited to small data objects.. However, there have been many restrictions on what has been allowed in the programming model with large objects, and the infrastructure led to an error prone implementation of large objects. For example, most string functions and operators are not allowed for LOBs, and there are also shortcomings with the text pointer functions in particular regard to insertions and deletions. (Text pointers are physical references used as row locators in earlier releases of SQL Server). Moreover, there is no efficient means for modifying one or more rows of a LOB at a time, and there is no support for LOB variables or copy functions for LOBs of any kind. These limitations, and others, result in LOBs being perceptibly different from other data types and therefore more difficult to use.

[0004] What is needed is a large object infrastructure where users/programmers can handle large values (data blocks) in the same way smaller values are handled, and thereby eliminate the user-perceptible differences in the handling of small and large values, in order to provide a single comprehensive programming model.

SUMMARY OF THE INVENTION

[0005] Various embodiments of the present invention are directed to systems and methods for providing an infrastructure that can be used for reimplementation of text, ntext, and image data types as new varchar(MAX), nvarchar(MAX), varbinary(MAX), and XML data types, as well as support large user-defined types (UDTs), and further enables support of additional new data types and programming model constructs. Several embodiments of the present invention are also directed to providing a data-handling infrastructure where users/programmers can handle large values (data objects) in the same way smaller values are handled, and thereby eliminate the difference between small and large values in order to provide a single comprehensive programming model.

[0006] These various embodiments utilize Blob Handles (BHs) which are an internal representation of a large value. BHs are immutable and stateless references to large data. The structure of a BH contains enough information to return an ILockBytes interface in order to provide access to the corresponding large data block, and a BH can also return information regarding its own lifetime description. These concepts and elements are described in more detail herein below.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] The foregoing summary, as well as the following detailed description of preferred embodiments, is better understood when read in conjunction with the appended drawings. For the purpose of illustrating the invention, there is shown in the drawings exemplary constructions of the invention; however, the invention is not limited to the specific methods and instrumentalities disclosed. In the drawings:

[0008] Fig. 1 is a block diagram representing a computer system in which aspects of the present invention may be incorporated;

[0009] Fig. 2 is schematic diagram representing a network in which aspects of the present invention may be incorporated;

[0010] Fig. 3A is a block diagram illustrating the general components of a BLOB Handle (BH) in various embodiments of the present invention.

[0011] Fig. 3B is a block diagram illustrating a BH wherein a value is entirely contained within said BH according to various embodiments of the present invention.

[0012] Fig. 4A is a block diagram illustrating an original BH and a second BH representing a virtual copy both pointing to the same data block for read-type operations.

[0013] Fig. 4B is a block diagram of the BHs from Fig. 4A wherein a write operation is called by either BH and the second BH is pointed to a newly created copy of the data block.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENTS

[0014] The inventive subject matter is described with specificity to meet statutory requirements. However, the description itself is not intended to limit the scope of this patent. Rather, the inventor has contemplated that the claimed subject matter might also be embodied in other ways, to include different steps or combinations of steps similar to the ones described in this document, in conjunction with other present or future technologies. Moreover, although the term "step" may be used herein to connote different elements of methods employed, the term should not be interpreted as implying any particular order among or between various steps herein disclosed unless and except when the order of individual steps is explicitly described.

Computer Environment

[0015] Numerous embodiments of the present invention may execute on a computer. Fig. 1 and the following discussion is intended to provide a brief general description of a suitable computing environment in which the invention may be implemented. Although not required, the invention will be described in the general context of computer executable instructions, such as program modules, being executed by a computer, such as a client workstation or a server. Generally, program modules include routines, programs, objects, components, data structures and the like that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand held devices, multi processor systems, microprocessor based or programmable consumer electronics, network PCs, minicomputers, mainframe computers and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0016] As shown in Fig. 1, an exemplary general purpose computing system includes a conventional personal computer 20 or the like, including a processing unit 21, a system memory 22, and a system bus 23 that couples various system components including the system memory to the processing unit 21. The system bus 23 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system 26 (BIOS), containing the basic routines that help to transfer information between elements within the personal computer 20, such as during start up, is stored in ROM 24. The personal computer 20 may further include a hard disk drive 27 for reading from and writing to a hard disk, not shown, a magnetic disk drive 28 for reading from or writing to a removable magnetic disk 29, and an optical disk drive 30 for reading from or writing to a removable optical disk 31 such as a CD ROM or other optical media. The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical drive interface 34, respectively. The drives and their associated computer readable media provide non volatile storage of computer readable instructions, data structures, program modules and other data for the personal computer 20. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 29 and a removable optical disk 31, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs) and the like may also be used in the exemplary operating environment.

[0017] A number of program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24 or RAM 25, including an operating system 35, one or more application programs 36, other program modules 37 and program data 38. A user may enter commands and information into the personal computer 20 through input devices such as a keyboard 40 and pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite disk, scanner or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor 47, personal computers typically include other peripheral output devices (not shown), such as speakers and printers. The exemplary system of Fig. 1 also includes a host adapter 55, Small Computer System Interface (SCSI) bus 56, and an external storage device 62 connected to the SCSI bus 56.

[0018] The personal computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 49. The remote computer 49 may be another personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the personal computer 20, although only a memory storage device 50 has been illustrated in Fig. 1. The logical connections depicted in Fig. 1 include a local area network (LAN) 51 and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise wide computer networks, intranets and the Internet.

[0019] When used in a LAN networking environment, the personal computer 20 is connected to the LAN 51 through a network interface or adapter 53. When used in a WAN networking environment, the personal computer 20 typically includes a modem 54 or other means for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used. Moreover, while it is envisioned that numerous embodiments of the present invention are particularly well-suited for computerized systems, nothing in this document is intended to limit the invention to such embodiments.

Networking Environment

[0020] Fig. 2 illustrates an exemplary network environment in which the present invention may be employed. Of course, actual network and database environments can be arranged in a variety of configurations; however, the exemplary environment shown here provides a framework for understanding the type of environment in which the present invention operates.

[0021] The network may include client computers 20a, a server computer 20b, data source computers 20c, and databases 70, 72a, and 72b. The client computers 20a and the data source computers 20c are in electronic communication with the server computer 20b via communications network 80, e.g., an Intranet. Client computers 20a and data source computers 20c are connected to the communications network by way of communications interfaces 82. Communications interfaces 82 can be any one of the well-known communications interfaces such as Ethernet connections, modem connections, and so on.

[0022] Server computer 20b provides management of database 70 by way of database server system software, described more fully below. As such, server 20b acts as a storehouse of data from a variety of data sources and provides that data to a variety of data consumers.

[0023] In the example of Fig. 2, data sources are provided by data source computers 20c. Data source computers 20c communicate data to server computer 20b via communications network 80, which may be a LAN, WAN, Intranet, Internet, or the like. Data source computers 20c store data locally in databases 72a, 72b, which may be relational database servers, excel spreadsheets, files, or the like. For example, database

72a shows data stored in tables 150, 152, and 154. The data provided by data sources 20c is combined and stored in a large database such as a data warehouse maintained by server 20b.

[0024] Client computers 20a that desire to use the data stored by server computer 20b can access the database 70 via communications network 80. Client computers 20a request the data by way of SQL queries (e.g., update, insert, and delete) on the data stored in database 70.

BLOB Handles

[0025] Various embodiments of the present invention are directed to BLOB Handles (BHs). A BH (along with its interface, IBlobHandle), is an internal representation of an immutable and stateless reference to a large object (LOB) (hereinafter a "data block"). Using BHs, all instances of LOBs in a system (including BLOBs, which are binary large objects) can be consolidated into a single representation. In essence, a BH is a reference to locate a data block, wherein the BH completely describes the data block. The BH itself can be described with (a) a pointer to the beginning of BH data and (b) the byte-length of the BH.

[0026] For example, in C++, a BH supports method invocation like regular C++ objects using a pointer to the beginning of the buffer. The structure of a BH contains enough information to return an ILockBytes interface in order to provide access to the corresponding data block, and some BHs may support a "delete" method which destroys associated storage. BHs can also return information regarding their own lifetime descriptions. Furthermore, for small values the BH will be able to contain the value within itself and thereby bypass the retrieval of an ILockBytes interface and the virtual method calls to read from a separate data block.

[0027] Fig. 3A is a block diagram illustrating the general components of a BH in various embodiments of the present invention. The BH 300 comprises (a) an offset 302 for a pointer 304 to the starting location of a corresponding data block 312 stored somewhere in memory or disk storage ("storage") 310, and (b) and a length 306 corresponding to the size of the data block 312 in storage 310. The BH further comprises a single-bit lifetime description (not shown).

[0028] Fig. 3B is a block diagram illustrating the same BH of Fig. 3A but wherein the offset 302 of said BH 300, instead of pointing to an externally-referenced LOB, points 304 to a smaller value 320 that is entirely contained within said BH 310.

<u>BH Infrastructure</u>

[0029] In the BH infrastructure, large value parameters are handled by using a copy of the BH (instead of a copy of the data block) to provide a copy on demand as an input parameters. This copied BH is passed as a pointer reference by using the BH aliasing mechanism provided by the BH infrastructure. However, if changes to the data block are necessary for one copy of the BH, then the large value is copied and the writing BH would point to this new copy alone; thus this copy functionality, by necessity, would generally occur for output parameters.

[0030] Fig. 4A is a block diagram illustrating an original BH 402 pointing 404 to a data block 412 and the creation of a second BH 402' that represents a virtual copy 412' of the data block 412 but which in fact points 404' to the same data block 412. However, as illustrated in Fig. 4B, if the data in the "copy" represented by second BH

402' (or, conversely, if the data represented by BH 402) is to be changed by a write operation, then data block 412 is truly copied to a new data block 412" and the second BH 402' will point 404' to this second data block 412".

[0031] However, for various embodiments, the BH infrastructure may also support "partial updates" to data blocks—that is, the ability to change only a portion of a large value without incurring a full copy of the entire data value—to preclude some of the necessity for copying the entire data block. In operation, a partial update ascertains (a) a replacement value corresponding to a "chunk" of a data block (a chunk being a subpart of said data block), (b) an "offset" for the beginning of the chunk in the data block, and (c) the byte-length of the chunk to be replaced (which also corresponds to the byte-length of the replacement value). Logically, this approach simply replaces the byte-length number of bytes at the offset position of the data block (which essentially defines the metes and bounds of the subject chunk) with the aforementioned replacement value having an identical byte-length. In alternative embodiments, the replacement value need not be the same byte-length as the chunk being replaced, and smaller or larger values can be put into the chunk using various methods (e.g., pointer-based methods) known and appreciated by those of skill in the art. Moreover, either of these techniques can be used with the virtual copy concept described earlier herein and illustrated in Figs. 4A and 4B.

[0032] Internally, partial updates are represented as a special form of column update (that can be mixed with other partial or regular updates) and, where possible, the change is pushed to disk storage modify only the specific portions of the B-tree required. However, in some cases, the system may choose to implement the partial update as a full copy for reasons of efficiency or robustness, and in other cases the update would not be performed partially but, instead, a full replacement is performed. The system implements partial updates as full copies (i.e. replacing the old value with the new) in the presence of updates that require a movement of the data row to a different physical location, for examples, updates to the partitioning or clustering key. The reason for this is that the movement of the row to a different location requires a full copy, and it is more efficient to complete the operation in one step. It should also be noted that the system will not reject as unfeasible update statements containing partial updates, but will instead choose to make a full copy implementation in the aforementioned cases.

[0033] Partial updates infrastructure also takes into account "Halloween protection" (i.e. when values being used as arguments in the UPDATE are also being updated themselves).

[0034] This partial update infrastructure represents a big advance over previous text pointer based technology. Given that partial updates are known operations to the query processor (QP), several features are now enabled that used to be restricted in the text pointer implementation, including but not limited to: index and indexed view maintenance, partitioning, replication, and constraint validation.

[0035] As part of the BH infrastructure for several embodiments of the present invention, the underlying storage infrastructure would also have the concept of a column in-row limit. This limit determines the size boundary at which a given value is chosen to be pushed off row as opposed to being kept in the main page. The smaller the in-row limit is, the sooner a column is pushed off row, and a larger limit has the opposite effect. As known and appreciated by those of skill in the art, this reflects a trade-off where small limits enable a larger number of rows to fit per page which, in turn, increases the row density and improves scan performance, but where the large value columns are not part of the scan. In this regard, the BH infrastructure allows an internally configurable in-row limit per type. Thus MAX V-Types can expose a given behavior based on a specific in-row limit, while UDTs or XML types can choose a different limit and achieve different density and scan performance characteristics more suitable to their size and structure.

<u>BH Methods and API</u>

[0036] For various embodiments of the present invention, the BH may support one or more of the following methods which are described in more detail later herein:

```
interface IBlobHandle
{
public:

      virtual const CBlobHandleProperties *PbhpGet() const
= 0;
            // Returns blob handle properties.

      virtual ILockBytesSS *PilbGetILockBytes(_IN_
IlbGetFlags flags) const = 0;
            // Use regular Release to deallocate
ILockBytesSS.
            // ILockBytesSS may have limited support for
AddRef (e.q., may not return updated ref count).

      virtual void Delete() const = 0;
            // Allows to destroy storage associated with
temp handle (if any)

      virtual BOOL FGetPointer(_OUT_ _OPT_ LPCVOID *ppv,
_OUT_ _OPT_ ULONG *pcbLength) const = 0;
            // If TRUE then returns pointer + length to
read-only data(provided out parameters are not NULL).
```

```
                // If FALSE then in-memory pointer is not
        available.

        };
```

Typical implementation of BH structure is inherited from an IBlobHandle interface with optional data appended in certain instances. Since a BH is self-contained and does not have embedded pointers to itself, no assumption are requited about the memory where it resides. Hence, to copy a BH, only a simple shallow copy is required. This allows the system to treat BHs the same way as regular columns are treated.

[0037] Although the BH can be used to locate a corresponding data block, it does not provide interfaces to manipulate the LOB data of the data block. Instead, for certain embodiments the PilbGetILockBytes method would be used to retrieve ILockBytesSS for data manipulation. The exact set of ILockBytesSS methods supported would vary on BH implementation but would typically falls into two categories: (i) read-only LOBs that only support read operations, such as a BH that represents data from a user table and which is typically read-only (with exceptions in the case of a partial update); and (ii) read-write LOBs that support both reading and updates to the data block, such as temporary BHs used to store temporary LOB values.

[0038] In certain embodiments, a delete method on BHs could be invoked in order to delete the LOB value associated with the BH. However, only some BHs would support this operation (e.g., temporary BHs) while others would not (e.g., a BH from user table). In any event, the delete operation would be non-failing, which is useful for handling cleanup to ensure that auto-destructors do not fail.

[0039] BHs in certain embodiments also support the FGetPointer method that, as an alternative to ILockBytesSS, could be used to return a pointer to and the length of the LOB data block, and thereby provide a faster means for accessing small read-only data. However, if the data block is not comprised of small read-only data, the method may return FALSE to indicate that the PilbGetILockBytes method should be used to access data instead.

[0040] For various embodiments, BH API methods except for Delete may throw exceptions.

BH Properties

[0041] In several embodiments the PbhpGet method returns CBlobHandleProperties which is an encapsulation of BH properties. Typically many BHs share common properties, and a comparison of properties via a pointer (CBlobHandleProperties *) is used to distinguish a BH's "flavor" (e.g., whether a BH points to data in a user table or instead points to a "custom" data block; BH Flavors are discussed in more detail later herein).

[0042] Several embodiments utilize a "lifetime" property for the length of time a BH reference is valid. For example, a BH may no longer be associated with an existing LOB value after some kinds of events. For certain embodiments, this property may simply be assigned one of two values, "query lifetime" or "row lifetime" (or other binary-equivalent designations). "Row lifetime" indicates the associated LOB values can be accessed only while a Query Execution (QE) iterator is processing a current row, and once the iterator proceeds to the next row the BH is no longer associated with the LOB value (and thus any further access is via the BH is prohibited). "Query lifetime," on the other hand, would indicate that the LOB values are valid until the end of the query (or, in other non-query cases, until the "BH Factory" is purged or shutdown, BH Factories being discussed in greater detail elsewhere herein).

BH Flavors

[0043] Several embodiments of the present invention comprise one or more "flavors" of BHs, said flavors comprising the following BH implementation variations:

- A "table" BHs that points to a data block in a user table (and would typically have a "query lifetime").
- A "temporary" BH created using a BH Factory (discussed elsewhere herein) for temporary utilization.
- An "inlined" BH that, for sufficiently small data blocks, contains the data block entirely within the BH body. For this flavor, the ILockBytesSS interface is also inside the BH body so the PilbGetILockBytes method will be very fast and efficient.
- An "alias" BH that redirects method calls to another BH, which is particularly useful for certain operation such as converting a BH that is larger than a TEMPBLOBHANDLE_SIZE (a predefined size limit) to a BH that is less than or equal to a TEMPBLOBHANDLE_SIZE (as described in more detail herein below).
- A "wrapper" BH that redirects access to an existing ILockBytesSS object. This is particular useful when used to "cast" an existing ILockBytes object from another source (e.g., from an OLEDB) into a BH.

An "orphan" BH that belongs to a specified filegroup and which can be linked to a user table at a later time. In operation, when a large value is streamed into the system, the large value can be copied directly into an orphan BH and, once the data is fully streamed, the BH is then made a part of the table being inserted to ("adopted") without need for additional copy. This kind of BH is thus used to minimize copies as well as improve performance of high performance bulk operations, such as BULK INSERT statement.

BH Maximum Size

[0044] For certain embodiments, assumptions about maximum buffer size are needed to contain the BH. For these embodiments, a QE would typically use the maximum size derived by a Query Optimizer (QO), although a very maximum BH size could also be defined as constant, e.g., const BLOBHANDLE_SIZE = 8000. However, a "temporary" BH could have a limit based on the processor architecture, e.g., defined as const TEMPBLOBHANDLE_SIZE = 0x40 for a 32-bit architecture. The main idea behind this approach is that most components would either access BHs by reference (and without allocating their own buffer) or use temporary BHs to spool the large value of a data block. Therefore it may be desirable to allocate small buffers (that is, of size TEMPBLOBHANDLE_SIZE) if the BH will be persisted. On the other hand, QE will typically use much larger BHs. However, the important consideration here is that almost any component can handle BHs of TEMPBLOBHANDLE_SIZE size, but if the BH is larger than this value and is visible to components outside Query Execution then it must only be accessed by its associated pointer. For example, if QE exposes large BHs to a

Language Processing and Execution component (LPE), the LPE typically cannot copy the large BH into its own buffer and thus must use a pointer to the QE buffer.

BH Factories

[0045] A BH Factory (BHF) is an object that creates new BHs. In general, there may be no common interface supported by all BF Factories, but there is be a common interface supported by "Temporary BH Factories" (TBHFs) whereby temporary data blocks are used to store large data for processing by one or many components. In the latter case, an IBlobHandleFactory pointer is passed from one component to another for continued processing, and the TBHF has optimizations that make processing of data blocks representing smaller values (e.g., less than 8K) more efficient (as described above). There is also a special BHF that supports parallel access from multiple threads (decribed in more detail below).

[0046] The following method creates a BH using a TBHF:

```
            // Function creates BH in specified buffer.
Buffer size is in/out parameter and returns created handle
size.
            virtual void CreateNewBlobHandle(IBlobHandle
*pbh, ULONG& cbBufferSize, CreateNewBlobHandleFlags flags)
= 0;
```

[0047] In certain embodiments, this method creates a new BH associated with an "empty" LOB value. Data can then be written into this BH using ILockBytesSS, and additional methods to close and release the BH (ILockBytesSS::Close and ILockBytesSS::Release) could be used.

[0048] In general, BHFs can service multiple threads, though the specific thread that creates the BHF owns that particular BHF and is the only thread allowed to cleanup the BHF when the time comes. BHFs can also create BHs in groups by tagging the member BH so created with a given group identification number. This enables the BHF to provide services for releasing all BHs associated with a given group with a single operation.

[0049] Depending on the BHF, the BH created by a BH may be backed by disk or by memory. Initially (and preferably) BHFs do not have specific disk storage associated with them, but in general storage is created on demand on first use or through an explicit request. In this regard, various embodiments of the present invention utilize one or more types of BHFs, including but not limited to the following:

- Fast BHFs: These BHFs create in-memory-only BHs which support smaller values by using memory backing instead of permanent disk storage. In general, the BH infrastructure would attempt to use a fast BHF when possible but would fall back on the slower disk backed storage if needed.
- Disk Backed BHFs: These BHFs create BHs that are backed by disk storage and can hold values arbitrarily large (disk space permitting).

The MAX specifier and MAX V-types

[0050] Using BHs, various embodiments of the present invention are also directed to the use of a MAX specifier for the varchar, nvarchar, and varbinary data types (together hereinafter the "V-types"). For example, table columns and TSQL variables may specify varchar(MAX), nvarchar(MAX), or varbinary(MAX) (together hereinafter the "MAX V-types") at definition/declaration time. In operation, MAX V-type columns and variables would use the "MAX" keyword as the specifier for the size of the type, and such values would be defined based on a size supportable by the system—e.g., varchar(MAX) != varchar(<a valid number>)—although the largest literal that can be used at declaration to specify the max size for a varchar or varbinary continues to be 8000 and 4000 for an nvarchar. (Normally varchar or varbinary columns have a maximum size of 8000, but the MAX keyword qualifies columns whose maximum size is equivalent to the highest supportable by the system, using the BH infrastructure.)

[0051] In various alternative embodiments, one or more of the following operations and functionality would then be supported for the MAX V-types: comparison; TSQL variables; concatenation; certain string functions; full SELECT statement support, including DISTINCT, ORDER BY, GROUP BY, aggregates, joins, subqueries, and so forth; CHECK constraints, RULEs and DEFAULTs (were defaults greater than 8K are no longer silently truncated as happens today with LOBs); being part of indexed views definitons; visibility on inserted/deleted tables on AFTER triggers; and data type conversions; allowed in FFO cursors; FETCH... INTO...@variable; SET ANSI_PADDING; SET CONCAT_NULL_YIELDS_NULL; Full Text Search; and CREATE TYPE (to replace sp_addtype), among other things. However, because of the real and substantial size differences, for some embodiments of the present invention subtle differences between MAX V-types and V-types would continue to exist, including but not limited to the following: sql_variant still cannot contain MAX V-types; MAX V-type columns cannot be specified as a key column in an index (although they are allowed to be used as index subkeys); and MAX V-types columns cannot be used as partitioning key columns. It is however allowed to specify a prefix of the column as the key of the index or indexed view. This can allow the system to automatically consider the index in order to quickly retrieve the rows that qualify certain kinds of predicates involving the MAX V-type column. For example, a predicate in the form [Varchar(MAX) column] = Value, can be automatically split by the system into Prefix([Varchar(MAX) column]) = Prefix(Value) AND [Varchar(MAX) column] = Value. The implied predicate over the prefix of the column can allow the usage of the index.

[0052] To ensure backward compatibility, when sending data downlevel to clients that do not support MAX V-types, the varchar(MAX), nvarchar(MAX), and varbinary(MAX) data types would be sent as LOB-type text, ntext, and image types respectively. Conversely, migration support for data receiving uplevel from clients in the form of LOB-type text, ntext, and image data would be transformed into the varchar(MAX), nvarchar(MAX), and varbinary(MAX) data types respectively.

Conclusion

[0053] The various systems, methods, and techniques described herein may be implemented with hardware or software or, where appropriate, with a combination of both. Thus, the methods and apparatus of the present invention, or certain aspects or portions thereof, may take the form of program code (i.e., instructions) embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. In the case of program code execution on programmable computers, the computer will generally include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device. One or more programs are preferably implemented in a high level procedural or object oriented programming language to communicate with a computer system. However, the program(s) can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language, and combined with hardware implementations.

[0054] The methods and apparatus of the present invention may also be embodied in the form of program code that is transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via any other form of transmission, wherein, when the program code is received and loaded into and executed by a machine, such as an EPROM, a gate array, a programmable logic device (PLD), a client computer, a video recorder or the like, the machine becomes an apparatus for practicing the invention. When implemented on a general-purpose processor, the program code combines with the processor to provide a unique apparatus that operates to perform the indexing functionality of the present invention.

[0055] While the present invention has been described in connection with the preferred embodiments of the various figures, it is to be understood that other similar embodiments may be used or modifications and additions may be made to the described embodiment for performing the same function of the present invention without deviating there from. For example, while exemplary embodiments of the invention are described in the context of digital devices emulating the functionality of personal computers, one skilled in the art will recognize that the present invention is not limited to such digital devices, as described in the present application may apply to any number of existing or emerging computing devices or environments, such as a gaming console, handheld computer, portable computer, etc. whether wired or wireless, and may be applied to any number of such computing devices connected via a communications network, and interacting across the network. Furthermore, it should be emphasized that a variety of computer platforms, including handheld device operating systems and other application specific hardware/software interface systems, are herein contemplated, especially as the number of wireless networked devices continues to proliferate. Therefore, the present invention should not be limited to any single embodiment, but rather construed in breadth and scope in accordance with the appended claims.

[0056] Finally, the disclosed embodiments described herein may be adapted for use in other processor architectures, computer-based systems, or system virtualizations, and such embodiments are expressly anticipated by the disclosures made herein and, thus, the present invention should not be limited to specific embodiments described herein but instead construed most broadly.